Pseudo-Exhaustive Verification of Rule Based Systems

D. Richard Kuhn[1], Dylan Yaga[1], Raghu N. Kacker[1], Yu Lei[2], Vincent Hu[1]

[1] National Institute of Standards and Technology
Gaithersburg, MD 20899, USA
{kuhn,dylan.yaga,raghu.kacker}@nist.gov

[2]Computer Science & Engineering
University of Texas at Arlington
Arlington, TX, USA
ylei@uta.edu

Abstract — **Rule-based systems are important in application domains such as artificial intelligence and business rule engines. When translated into an implementation, simple expressions in rules may map to a large body of code that requires testing. We show how rule-based systems may be tested efficiently, using combinatorial methods and a constraint solver in a test method that is *pseudo-exhaustive*, which we define as exhaustive testing of all combinations of variable values on which a decision is dependent. The method has been implemented in a tool that can be applied to testing and verification for a wide range of applications.**

Keywords — combinatorial testing; constraint solvers; formal methods; t-way testing; rule-based systems; test automation

I. INTRODUCTION

Rule-based systems have been important in a variety of application domains for many years. Some of the earliest artificial intelligence systems (AI) were designed to evaluate large rule sets, and this approach continues to be important for AI. In other domains, business rule engines automate complex enterprise resource planning (ERP) problems [1]. The terms used in rules may be expressed as Boolean (dichotomous) or relational conditions on inputs, values from databases, and environmental conditions such as time of day. Thus even a rule that contains only a few simple conditionals may invoke significant processing involved in computing the values used in the rule conditions. A rule-based system must work for any set of inputs, and can be implemented with a wide variety of rule engines. For example, JBoss, Oracle Policy Automation, OpenRules, Drools, IBM ODM, and many other tools exist to process rules supplied by users. But as with conventional software, exhaustive testing is nearly always intractable. This paper generalizes a practical method developed for testing access control systems [2], and introduces a tool that implements this method.

The approach to testing rule-based systems is *pseudo-exhaustive*, which we define as exhaustive testing of all combinations of variable values on which a decision is dependent. This approach is analogous to pseudo-exhaustive methods for testing combinational circuits [3], where the verification problem is reduced by exhaustively testing only the subset of inputs on which an output is dependent, or by partitioning the circuit and exhaustively testing each segment. The general concept of exhaustively testing subsets of variable values on which a decision is dependent is applied here to rule-based systems by transforming rule conditions to disjunctive normal form, then considering each term separately [2].

DOI reference number: 10.18293/SEKE2018-072

Testing a rule-based system requires showing that the rules as specified, P, are correctly implemented. The implementation P' must be shown to produce the same response as P for any combination of input values used in rules. That is, for input values $x_1,\ldots, x_n$, $P'(x_1,\ldots, x_n) = P(x_1,\ldots, x_n)$. Positive testing to show that a rule produces a specified result is easy: instantiate conditions to true for each antecedent associated with the result and verify that the system returns the designated result. Negative testing, showing that no combination of input values will produce the same result when it should not, is much more difficult. With n Boolean variables there are 2^n possible combinations of variables. For example, it would not be unusual to have 50 Boolean variables, resulting in $2^{50} \approx 10^{15}$ combinations, which would appear to make full negative testing intractable. In this paper, we show how combinatorial methods can be used to make this testing problem practical, given assumptions that apply to many or most rule-based systems.

II. TEST CONSTRUCTION

We describe the derivation of complete test cases from rules converted to k-DNF structure (disjunctive normal form where no term contains more than k literals, and a *term* is a conjunction of one or more literals within the disjunction), using a constraint solver and a covering array generator. Two arrays are constructed for each possible rule consequent, such that every test in each array should produce the same result, with variations indicating an error. The method may be applied to rule systems with multiple outputs, where outputs are either discrete values or are defined by a predicate or expression with a Boolean result.

Rules are assumed to be given as expressions made up of variables with logical connectives in an antecedent, with a consequent given as a discrete value or simple predicate, structured as shown below where R_i are predicates evaluating the values of one or more variables, and $result_i$ is the result expected when conditions of R_i evaluate to true:

$$(R_1 \rightarrow result_1)\ (R_2 \rightarrow result_2)\ \ldots\ (R_m \rightarrow result_m)$$

else → *default*

which is equivalent to:

$(R_1 \rightarrow result_1)\ (R_2 \rightarrow result_2)\ \ldots\ (R_m \rightarrow result_m)$

$(\sim R_1)\ (\sim R_2)\ldots\ (\sim R_m) \rightarrow$ *default*

Each R_i may include multiple variables, conditions, and logical connectives. It is required that the rule antecedents R_i are mutually exclusive, i.e., for any set of input variable values, only one antecedent will be matched. We believe this requirement is not overly restrictive, as in most applications it would be an error for matches of more than one rule. (It would be possible to use the constraint solver to check that rule antecedents are mutually exclusive, but this feature has not been implemented.)

Example 1: Suppose we have a rule set as shown below:

```
if (a && (c && !d ||e))  R1;
else if (!a && b && !c)  R2;
else exit();
```

This code can be mapped to the following expression (note second line is "else", i.e., negation of predicates for R1 and R2):

$(a(c\bar{d}+e) \rightarrow R_1)\ (\bar{a}\ b\ \bar{c} \rightarrow R_2)$

$((\sim(a(c\bar{d}+e)))(\sim(\bar{a}\ b\ \bar{c})) \rightarrow exit)$

Literals can be conditions, such as age>18, or Boolean variables such as employee (yes, no), but the structure will be a series of expressions specifying subsets of conditions that produce each result, followed by a default rule when none of the attribute expressions have been instantiated to true.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
2	0	0	1	0	1	1	1	1	1	0	0	0	0	0	1
3	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
4	0	1	1	1	1	0	0	1	0	0	0	1	0	0	1
5	1	0	0	0	1	0	0	1	0	0	1	1	1	1	0
6	1	0	1	1	0	1	1	1	0	1	0	1	1	1	0
7	1	1	0	1	1	1	0	0	0	1	0	0	1	0	1
8	1	1	1	0	0	0	1	1	1	1	1	0	1	1	1
9	1	0	0	0	0	1	1	0	1	0	0	1	1	1	1
10	0	1	1	1	0	1	1	0	1	0	1	1	0	1	0
11	0	1	0	0	1	0	1	0	0	0	1	0	1	1	1
12	1	0	1	1	1	0	0	0	1	0	1	0	1	0	0
13	1	0	0	1	1	0	1	1	1	1	1	1	0	1	1
14	1	0	1	0	0	1	0	0	0	0	1	1	0	0	1
15	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0
16	0	1	1	0	1	0	0	0	1	1	0	1	1	1	1
17	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0
18	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0
19	0	1	0	0	1	0	0	1	0	1	0	0	0	0	0
20	1	1	1	1	0	0	1	0	0	1	1	1	0	1	1
21	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0
22	0	1	1	1	0	1	1	1	0	1	0	0	1	1	1

Figure 1. 3-way covering array of 15 boolean parameters

To make testing tractable, we use combinatorial methods [2][4]. To see the advantages of a combinatorial approach, refer to Fig. 1, which shows a covering array of 15 boolean variables. A covering array is an $N \times k$ array of N rows and k variables. In every $N \times t$ subarray, each t-tuple occurs at least once. In software testing, each row of the covering array represents a test, with one column for each parameter that is varied in testing. For example, Fig. 1 shows a complete 3-way covering array that includes all 3-way combinations of binary values for 15 parameters in only 22 tests. The size of a t-way covering array of n variables with v values each is proportional to $v^t \log n$ [6][7]. For Example 1, with five attributes and two possible decisions for each attribute, there are $2^5 = 32$ possible rule instantiations. However, a covering array of all 3-way combinations contains only 12 rows. The number of variables for which all settings are guaranteed to be covered in a covering array is referred to as the *strength*; a 3-way array is of strength 3. We use covering arrays of variables from rules that have been converted to k-DNF form. For example, $abc + de$ contains two terms, one with three literals and one with two, so the expression is in 3-DNF form. The covering array does not contain all possible input configurations, but it will contain all k-way combinations of variable values. Where an expression is in k-DNF, any term containing k literals that is resolved to true will clearly result in the full expression being evaluated to true. For example, an access control rule in 2-DNF form could be: "if employee && US_citizen || auditor then *grant*". This rule contains one term of two attributes and one term of one attribute, so it is 2-DNF. Because a covering array of strength k contains every possible setting of all k-tuples and i-tuples for $i < k$, it contains every combination of values of any k literals.

As noted in the Introduction, we exhaustively test all combinations of values on which a decision is dependent. For the example above, the decision grant depends on either of two terms being true: employee && US_citizen or auditor. Any other setting of these three variables should result in deny. A truth table of all eight possible settings of these three variables would allow exhaustive testing of this set of rules. In general, exhaustive testing is intractable for nearly all applications, but note that at most two variables are required to produce a grant result. So if we test all 2-way combinations of settings of the input variables, we have achieved exhaustive testing of all combinations of variable values on which a decision is dependent, since no decision depends on more than two variables. (Later in the paper we show how this approach scales up to larger problems, and address the effectiveness for detecting errors when implemented rules contain more variables than are included in specified rules.)

Covering array generation tools, such as ACTS [4][6], make it possible to include constraints that prevent inclusion of variable combinations that meet criteria specified in a first order logic style syntax. For example, if we are testing applications that run on various combinations of operating systems and browsers, we may include a constraint such as 'OS = "Linux" => browser != "IE"'. Constraints are typically used in situations such as this, where certain combinations do not occur in practice or are physically impossible, and therefore should not be included in tests. Modern constraint solvers such as Choco [8] and Z3 [9] make it possible to process very complex constraint sets, converting logic expressions into combinations that are invalid and can be avoided in the final array.

Method: Let R = rule antecedents (left side of an implication rule such as p in $p \rightarrow q$) of one or more rules being tested in k-DNF, and T_i are terms (conjuncts of one or more variables or

terms) in R. We designate the result/consequent of the rule being tested as (+), and any other possible result as (-). For the example included in Example 1, terms T_i of R_1 would be $ac\bar{d}$, and ae, and R_1 would be designated as (+) and R_2 or exit() designated as (-), for this test.

Positive testing: Generate a test set PTEST for which every test should produce a particular response. It must be shown that for all possible inputs, where some combination of k input values matches a (+) condition, a (+) result is returned. Construct test set PTEST = $\{P_{TESTi}\}$ with one test for each term T_i of R as follows: $P_{TESTi} = T_i(\bigwedge_{j\neq i} \sim T_j)$

The construction ensures that each term in P is verified to independently produce the expected response for that rule. Negating each term T_j, $i \neq j$, prevents masking of a fault in the presence of other combinations that would return the same result. For example, if a rule condition is $ab + cd \rightarrow R_1$, inputs of 1100, 1101, 1110 could be used for testing $ab \rightarrow R_1$. However, input 1111 would not detect the fault if the system ignores variable a or b, because the condition cd would cause a result of R_1, and no other predicates in the rule would be evaluated. One such test is required for each term in a rule, so for m rules with an average of p terms each, the number of tests required is proportional to mp.

Negative testing: Generate a test set NTEST for which every test should produce a response other than the result designated by the rule being tested. It must be shown that for all possible inputs, where no combination of k input values matches a rule, an alternative result is returned.

> NTEST = covering array of strength k, for the set of variables in all rules, with constraints specified by $\sim R_i$.

Note that the structure of the rule evaluation makes it possible to use a covering array for NTEST, compressing a large number of test conditions into a few tests. Converted to k-DNF, each rule antecedent includes a sequence of conditions that are each sufficient to trigger the specified result. Because rule antecedents are mutually exclusive, masking of one combination by another can only occur for NTEST when a test produces a negative response, i.e., a response that is not a consequent of the rule instantiated in PTEST. In such a case, an error has been discovered, which can be repaired before running the test set again. Since NTEST is a covering array, the number of tests will be proportional to $v^k \log n$, for v values per variable (normally v=2 since most will be Boolean conditions), and n variables.

Rule antecedents are assumed to be mutually exclusive (to prevent masking as discussed above), but we allow for cases where multiple rules may have the same consequent (result). In such cases, rule antecedents are combined to produce the set of conjuncts used in generating PTEST and NTEST arrays. For example, if two rules are $R_1 \rightarrow Q_1$ and $R_2 \rightarrow Q_1$, then k-DNF terms for PTEST are produced from $(R_1 + R_2)$ and constraints for NTEST are given by $\sim(R_1 + R_2)$. For m rules with the same consequent (result), the number of tests is multiplied by the constant m.

Example 2: Table I gives a set of Boolean variables a through e, where each row defines values for the variables that determine an access control decision, either *grant* (+) or *deny* (-). Thus a covering array for the antecedent R of a rule in 3-DNF such as $(ac\bar{d} + \bar{a}b\bar{c} \rightarrow grant)$ is given in Table 1. The total number of 3-way combinations covered is the number of settings of three binary variables multiplied by the number of ways of choosing three variables from five, i.e., $2^3 \binom{5}{3} = 80$.

TABLE I. 3-WAY COVERING ARRAY

	a	b	c	d	e
1	0	0	0	0	0
2	0	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	1
6	1	0	1	0	0
7	1	1	0	0	1
8	1	1	1	1	0
9	1	1	0	0	0
10	0	0	1	1	0
11	0	0	0	0	1
12	1	1	1	1	1

TABLE II. 3-WAY COVERING ARRAY WITH CONSTRAINT ~R

	a	b	c	d	e
1	0	0	0	0	0
2	0	0	1	1	1
3	0	1	1	0	0
4	1	0	0	1	0
5	1	0	1	1	0
6	1	1	0	0	1
7	1	1	1	1	1
8	0	0	1	0	1
9	1	1	0	1	0
10	0	0	0	1	1
11	1	0	0	0	0
12	0	1	1	1	0
13	1	0	0	0	1
14	0	1	1	0	1

Table II shows a covering array for this set of variables generated using $\sim R$ as a constraint. That is, the two terms of the rule, $ac\bar{d}$ and $\bar{a}b\bar{c}$, have been excluded from the array, but all other 1-, 2-, and 3-way combinations can be found in the array. Because $ac\bar{d}$ and $\bar{a}b\bar{c}$ are the only conditions under which access should be granted, the array in Table II should result in a *deny* response from the system for every test. Collectively, tests include all 78 3-way settings of variables that will not instantiate the access control rule to *true*.

III. FAULT DETECTION PROPERTIES

Now consider the faults that this method can detect. Suppose that some combination of variables exists that produces a different response than required by the rule set P, for example because of errors in code that instantiates variable values. Tests contained in PTEST and NTEST will detect a large class of

missing terms, added terms, or altered terms containing k or fewer variables. In this section we analyze faults that will be detected, and the underlying conditions in these faults. Table III illustrates the fault types and detection conditions for each.

TABLE III. EXAMPLE FAULTS AND DETECTION CONDITIONS.

	Term	C=correct term	F=faulty term	PTEST detect condition	NTEST detect condition	notes
1	**missing**	*abc*	--	*abc*	*none*	
2	**added**	--	*ab*	*none*	*ab*	
3		*abc*	$\bar{a}b$	*none*	$\bar{a}bc$, $\bar{a}b\bar{c}$	
4		*abc*	*ab*	*none*	$ab\bar{c}$	
5		*ab*	*abc*	--	--	*no fault*
6	**altered**	*abc*	$ab\bar{c}$	*abc*	$ab\bar{c}$	
7		*abc*	*ab*	*none*	$ab\bar{c}$	
8		*abc*	$\bar{a}b$	*abc*	$\bar{a}bc$, $\bar{a}b\bar{c}$	

k-DNF detection property: It is shown in [2] that collectively, tests from PTEST and NTEST will detect faults introduced by added, deleted, or altered terms with up to k variables. We can also show [2] that if more than k attributes are included in the altered term, some faults are still detected. Specifically, where a correct term has more than k variables and is not a subset of a faulty term, the fault will be detected. If a correct term is a subset of a faulty term in this case, some faults will be detected.

IV. SOFTWARE TOOL

The prototype research tool, Pseudo-Exhaustive Verifier (PEV), was developed in Java, and utilizes several open source external Java libraries. The software is packaged as a Java Archive (.jar) file which is directly executable as a Graphical User Interface (GUI), or can be run as a Command Line Interface (CLI) from a terminal.

The PEV software has been designed to accept rule sets comprised of Boolean variables, Boolean operators and relational expressions, implementing the algorithm described in Sect. II. The software parses the rule set, converts to Disjunctive Normal Form (DNF), inverts the DNF rule set, solves for positive conditions, and uses NIST's Automated Combinatorial Testing for Software (ACTS) tool [6] to compute a covering array for negative conditions.

Algorithm implementation: PEV utilizes several publicly available Java Archive libraries to generate test arrays. Transforming the input Boolean rule set to DNF is done using *jbool_expressions* [10], and the Choco constraint solver [11] is used for resolving relational statements.

Parsing: Parsing is a critical step of the PEV software, which occurs before any testing is performed. Since the software needs to accept input from the user, any input must be modified and sanitized prior to use, to ensure compatibility with the various APIs used, as well as to catch any syntactical problems prior to testing... The parser strips extraneous whitespace, and then normalizes Boolean operators (`&&, &, ||, |, !, ~`), and attempts to match open and closing parenthesis. This sanitization ensures compatibility with the various APIs used throughout the software, and catches any syntactical problems prior to testing.

The software is not restricted to Boolean expressions, and has initial support for relational expressions (e.g., b < 3;). Note that a semicolon is used to identify a relational expression. During parsing, PEV will locate numeric relational expressions and replace them with temporary Boolean variables. After the replacement, the rule set is processed as normal. The relational values are solved at a later step and the results are recorded.

Once the initial input rule set is parsed, the software will convert it to Disjunctive Normal Form (DNF) to be tested. The user will be presented with a breakdown of the DNF rule set (split on the OR statements), each part of which is a positive condition that needs to be solved. Additionally, the user can set minimum and maximum values for any relational variable found in the rule set.

Solve for positive conditions: Each individual expression between OR operators is an expression that, once solved, will produce one positive condition. These expressions represent the only possible positive conditions for the original rule set – so it is possible to produce exhaustive positive conditions.

Consider Fig. 1, with the original input rule set:

```
emp & age>18; & (fa | emt | med) | b<3;
```

Converted to DNF, this is:

```
((age > 18; & emp & emt) | (age > 18; & emp
& fa) | (age > 18; & emp & med) | b < 3;)
```

Splitting on the OR operators, there are four individual expressions for the positive conditions (replacing relational expressions with temporary Boolean variables `tmp0 = age > 18; and tmp1 = b < 3;`):

- `tmp0 & emp & emt`
- `tmp0 & emp & fa`
- `tmp0 & emp & med`
- `tmp1`

To solve these expressions, any variable present is evaluated with the following rules, as shown in Table V:

- Non-negated variables evaluate to true
- Negated variables evaluate to false
- Variables not present evaluate to false

Solve for negative conditions: Depending on the complexity of the input rule set, it may not be feasible to produce exhaustive negative condition output combinations. By utilizing combinatorial test methods, it is possible to generate covering arrays of sufficient strength to have good test coverage. The method for producing negative conditions can be found by generating the full covering array for all the unique Boolean variables within the rule set, and using the DNF rule set as a constraint – which will remove the positive conditions from the resulting output.

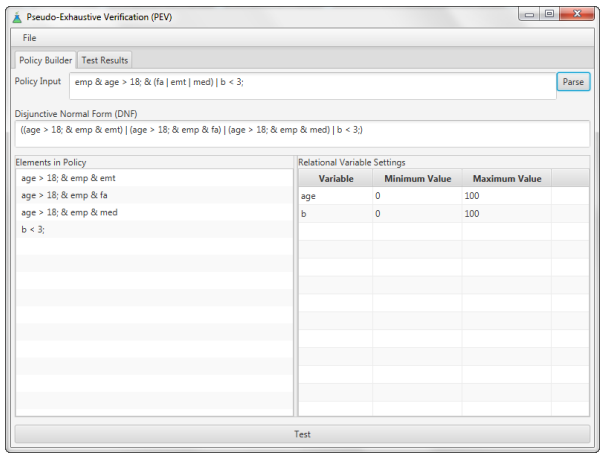

Figure 1. PEV software, after initial rule set parsed

TABLE V - SOLVED POSITIVE CONDITIONS

Expression	tmp0	tmp1	emp	emt	fa	med
tmp0 & emp & emt	1	0	1	1	0	0
tmp0 & emp & fa	1	0	1	0	1	0
tmp0 & emp & med	1	0	1	0	0	1
tmp1	0	1	0	0	0	0

A covering array for all negative conditions is computed as described in Sect. II. To perform this task, PEV creates an internal instance of the ACTS software, and passes a list of the unique Boolean variables from the rule set (including temporary Boolean replacements for relational expressions). The next step is to add the DNF rule set as a constraint to the system – so that the positive conditions are not included as negative results. Finally, the *k*-way combination is dynamically set after the *k*-DNF transform, which finds the conjunction with the largest combination of Boolean variables. In this example, the value of 3 is set (Table VI). PEV currently supports $k = 2..6$, because ACTS is used as the covering array generator, but there is no inherent limit to 6-way combinations and the method could support $k > 6$.

Solve for relational expressions using the Choco Expression Parser and the Choco Constraint Solver.

Relational Expression Formatting: The general format is:

```
Variable OPERATOR Integer_Value; Or
Integer_Value OPERATOR Variable;
```

Every relational expression must end with a semicolon (;), and two or more relational expressions in a row (without Boolean operators between them) will be replaced with one temporary Boolean variable during parsing. An example is shown in Table VII.

After being extracted and replaced by temporary Boolean variables, and the Positive/Negative conditions are found, an instance of Choco Expression Parser is created, and the relational expressions are passed as parameters. The minimum and maximum range for the expression to test against must be set – the PEV GUI includes a section which will allow the adjustment of every relational variable min and max values (default set to 0 to 100). These values can be adjusted prior to testing the rule set so that a customized range can be found. The solutions to the solved expressions are then placed into the results where appropriate.

TABLE IV. SOLVED NEGATIVE CONDITIONS

tmp0	tmp1	emp	emt	fa	med
1	0	1	0	0	0
1	0	0	1	1	1
0	0	1	1	1	0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	1	0	1	1
1	0	0	0	1	0
0	0	1	1	0	1
1	0	0	1	0	1
0	0	0	0	1	0
1	0	0	0	0	1
1	0	0	1	0	0

TABLE V. INPUT POLICIES AND RESULTING PARSED RULE SET

Input Rule set	Parsed Rule set
a > 10; 20 < b; || n	tmp0 || n
a > 10; || 20 < b; || n	tmp0 || tmp1 || n

Results: Once testing completes, PEV displays usage metrics and parameters which will result in positive conditions, and the covering array for negative conditions. At this point, the results can be saved as a comma separated value (.csv) file.

V. V. TEST SET SIZE AND PRACTICAL IMPLICATIONS

The process scales easily to systems with a large number of variables and rules. Because the number of rows in a covering array grows only with log *n* for *n* variables at a given number of values, a large increase in the number of variables requires only a few additional tests.

The most significant limitation for this approach occurs where terms in rules contain a large number of values per variable. Because the number of rows of a covering array increases with v^k, for *v* variable values, if terms in the rules have more than 10 to 12 values, it may not be practical to generate covering arrays. However, a large number of tests is not a barrier, because the structure of the solution resolves the oracle problem by ensuring that every test in PTEST should produce a response of (+) and every test in NTEST should produce a response of (-). Consequently, tests can be fully automated, making it possible to execute a large test set.

VI. VI. RELATED WORK

This paper generalizes a method developed originally for testing attribute-based access control systems [2], which had been incorporated into the Access Control Policy Testing tool ACPT [12]. The generalized method and new tool, PEV, were developed to make the method useful in development and testing for a wider range of applications. Pseudo-exhaustive test methods for circuit testing have an extensive history of application [1]. While our method is not derived from these earlier approaches, it shares the basic notion of determining dependencies, partitioning according to these dependencies, and

testing exhaustively the inputs on which an output is dependent. We have previously applied this notion to software testing in a more general form, using the observation that faults depend on a small number of inputs, by covering all 2-way to 6-way combinations of inputs [13]. This earlier work generated a test oracle using a model checker with a formal specification of a system, instantiated with inputs from a covering array.

Relatively little work has been published on testing specifically for rule-based systems. Dalal et al. [15] describe a case study of a rule-based system in an evaluation of model based testing, including the use of the combinatorial testing tool AETG. However, their testing considered only high level properties, such as whether updates correlated with the assignment of jobs during a working day. That is, no tests were generated from the rules. Rule based systems have also been used in a number of studies of test data generation [16][17], but used rules in generating tests for other software, rather than testing the rule-based systems themselves.

Among automated test generation systems, PEV falls into the class of tools with a specified test oracle, using the taxonomy of Barr et al. [14], because system rules serve as a specification of system behavior. Many such systems have been developed. The test oracles used in those systems were designed to answer the question "For a given set of inputs and initial state, what is the system output?", using a formal spec of some kind. Given such an oracle, test inputs must also be provided. Our method differs from these in that we address a narrower class of systems, but trade this limitation for complete coverage of inputs up to *k*-way combinations, providing testing that is pseudo-exhaustive, i.e., exhaustive for all subsets of inputs on which a rule result is dependent.

VII. Conclusions

Rule-based systems are used extensively in applications such as enterprise resource planning and machine learning [20]. If rules contain at most *k* Boolean variables per conjunction, for an expression in *k*-DNF, then a *k*-way covering array can test all possible settings of such terms. Thus for any possible combination of *n* inputs, only *k* ($k < n$) matter in determining the truth of the expression. In most applications, the number of conditions in conjunction will be small, even though the number of rules may be very high, possibly several hundred or even into thousands. The number of rows in a *k*-way covering array of Boolean variables is proportional to $2^k \log n$, and the ACTS covering array generator used in PEV produces arrays up to 6-way. Therefore PEV can efficiently process thousands of conditions or rules with up to six conditions per conjunction, sufficient for practical use.

The method described here was initially used in access control policy testing [2], and PEV has extended its applicability to a broader range of potential use. We are also considering methods to improve the efficiency of the PEV tool, including use of SAT solvers for generating covering arrays [18][19]. It may be possible to integrate the methods described in this paper with SAT-solver based covering array generation, to produce more compact arrays.

To make the tool more useful for practical application, features to allow import and export from common rule system formats, or decision table structures, may be helpful. We plan to investigate the possibilities depending on interest from users. We have received inquiries regarding compatibility with commercial tools, which could be considered for further development. Thus far, the major interest for this test method is for business rule systems, but it could be applied to traditional expert system applications as well.

Note: *Identification of products does not imply endorsement by NIST, nor that products identified are necessarily the best available for the purpose.*

References

[1] Lu, R., & Sadiq, S. A survey of comparative business process modeling approaches. In *Intl Conf on Business Information Systems* (pp. 82-94). Springer, 2007.

[2] Kuhn, D. R., Hu, V., Ferraiolo, D. F., Kacker, R. N., & Lei, Y. (2016, April). Pseudo-exhaustive testing of attribute based access control rules. In *Software Testing, Verification and Validation Workshops (ICSTW), 2016 IEEE Ninth International Conference on* (pp. 51-58).

[3] McCluskey, E. J. (1984). Verification Testing: A Pseudoexhaustive Test Technique. *Computers, IEEE Transactions on*, *100*(6), 541-546.

[4] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010). SP 800-142. Practical Combinatorial Testing, NIST, Gaithersburg, MD 20899

[5] ACTS Home Page, http:// csrc.nist.gov/acts/

[6] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: A general strategy for t-way software testing. *14th intl conference on the engineering of computer-based systems*, 2007, pp 549–556

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach toTesting Based on Combinatorial Design," *IEEE Trans. Software Eng.*, 23(7):437-444,1997.

[8] Jussien, N., Rochart, G., & Lorca, X. (2008). Choco: an open source java constraint programming library. *Open-Source Software for Integer and Contraint Programming (OSSICP'08)* (pp. 1-10).

[9] De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337-340). Springer Berlin Heidelberg.

[10] https://github.com/bpodgursky/jbool_expressions

[11] https://github.com/kaktus40/choco-exppar
http://www.choco-solver.org/

[12] ACPT Home Page, http://csrc.nist.gov/groups/SNS/acpt/ access_control_policy_testing.html

[13] D. R. Kuhn, V. Okun, Pseudo-exhaustive Testing For Software, 30th NASA/IEEE Software Engineering Workshop, April 25-27, 2006

[14] Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., & Yoo, S. (2015). The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, *41*(5), 507-525.

[15] Dalal, S. R., Jain, A., Karunanithi, N., Leaton, J. M., Lott, C. M., Patton, G. C., & Horowitz, B. M. (1999, May). Model-based testing in practice. *21st Intl Conf on Software Eng.* (pp. 285-294). ACM.

[16] Deason, W. H., Brown, D. B., Chang, K. H., & Cross, J. H. (1991). A rule-based software test data generator. *IEEE transactions on Knowledge and Data Engineering*, *3*(1), 108-117.

[17] Edvardsson, J. A survey on automatic test data generation. *2nd Conference on Computer Science and Engineering* (pp. 21-28) 1999.

[18] Lopez-Escogido D, Torres-Jimenez J, Rodriguez-Tello E, Rangel-Valdez N. Strength two covering arrays construction using a sat representation. InMICAI 2008: Advances in Artificial Intelligence 2008 Oct 27 (pp. 44-53). Springer Berlin Heidelberg.

[19] Banbara M, Matsunaka H, Tamura N, Inoue K. Generating combinatorial test cases by efficient SAT encodings suitable for CDCL SAT solvers. InLogic for Programming, Artificial Intelligence, and Reasoning 2010 Oct 10 (pp. 112-126). Springer.

[20] Lee, C. C. (1991). A self-learning rule-based controller employing approximate reasoning and neural net concepts. *International Journal of Intelligent Systems*, *6*(1), 71-93.